2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT) | 979-8-3503-4649-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/DCOSS-IOT58021.2023.00026

(POSTER) Insights from Executing TinyML Models on Smartphones and Microcontrollers

Harman M. Singh, Shrishailya Agashe, Shreyans Jain,
Surjya Ghosh, Aditya Challa, Sravan Danda, Sougata Sen
Dept. of Computer Science & Information Systems, BITS Pilani K K Birla Goa Campus, India

***Abstract*—In this paper, we empirically compare the system-level performances of executing a machine learning task on a resource-constrained IoT device and compare its performance to offloading the task to a more capable device, a smartphone. Our results indicate that although the resource-constrained device cannot run complex machine learning models, they can provide reasonable accuracy using similar models that can load on their memory. For running simpler models on the microcontroller, the best case accuracy for the machine learning task was 94.32% (SD: 1.7%). Furthermore, local computation resulted in almost 50% lesser current draw as compared to the offloading. These observations make a case for adopting an adaptive approach to ensure that applications meet the energy-accuracy-latency balance.**

I. Introduction

We, in this paper, perform a trade-off analysis between running machine learning tasks in situ, on a microcontroller (a commonly used IoT edge device) versus offloading the task to a co-located, resource-rich device. Specifically, we use deep learning-based object detection as the specific use case for comparing the performance of offloading the task to a connected smartphone, as compared to performing the object detection task on the microcontroller. We selected the object detection task as it is extensively used in various domains such as navigation, autonomous driving, and video surveillance [1]. We use TensorFlow to create the deep learning models, and the TensorFlow Lite (TFLite) framework[1] to convert the TF models. We use multiple variations of the mobile device-friendly MobileNets architecture as the deep learning models' architecture [2], [3]. Currently, only some microcontrollers support running TFLite models, one among which is the ESP32 microcontroller family.[2] We use an ESP32 microcontroller-based edge device to either perform the in situ task or to offload the task to a nearby smartphone.

Overall Goals: Thus, this paper aims to study the difference in performance (as a measure of accuracy, latency, and energy tradeoffs) when computation is performed on a low-powered microcontroller, as compared to performing the same task on a smartphone.

II. Experimental Setting

Devices chosen and connection strategy: We use an ESP-EYE development board to run various machine learning tasks [4]. The ESP-EYE development board consists of an ESP32 microcontroller, 8 MiB PSRAM, 4 MiB flash memory, 2 Megapixel camera, and Bluetooth Low Energy (BLE) and Wi-Fi modules. The ESP-EYE in the experiment was connected to an Android-based Samsung Galaxy A31 smartphone via Wi-Fi. In our setup, the ESP-EYE acted as the access point; the smartphone connected to this access point to ensure the connection was not routed via an external router.

Deep Learning approach: Deep convolution networks are known to extract features from images that can be used for object detection [2], [5]. The MobileNet architecture has been designed for low-resource environments, making it our preferred choice. Transfer learning is a commonly employed approach to reduce a network's training time and large data requirements. Many pre-trained MobileNets models are available online [6], which makes transfer learning a suitable training method. We applied transfer learning on both MobileNets V1 [2] and MobileNet V2 [3] architectures.

All models were retrained on the same dataset containing five object classes: bicycle, car, chair, person, and table. We used 200 images for retraining each class, along with data augmentation, and trained six different versions of MobileNets –three based on MobileNets

[1] https://www.tensorflow.org/lite, Accessed: 02/20/2023
[2] www.tensorflow.org/lite/microcontrollers, Accessed: 02/20/2023

2325-2944/23/$31.00 ©2023 IEEE
DOI 10.1109/DCOSS-IoT58021.2023.00026

V1 ($\alpha \in \{0.1, 0.2, 0.25\}$), and three based on MobileNets V2 ($\alpha \in \{0.05, 0.1, 0.35\}$). α is the width multiplier. For each version, we developed one quantized, and one unquantized model. Each model had an input shape of (96, 96, 3). These values signify the width, height, and the number of colors of the image, respectively.

The ESP and Smartphone Applications: We developed two microcontroller-based applications and one smartphone-based application. The **microcontroller-based applications** were developed using the ESP-IDF (Espressif IoT Development Framework). The first application allowed capturing images using the ESP-EYE and locally running the inference on the microcontroller. This application supported both quantized (8-bit int) and unquantized (32-bit float) models. The second ESP-based application enabled capturing images and offloading the inference task onto the smartphone via Wi-Fi. We experimented with various image-capturing rates on the ESP. The **smartphone application** obtained images from the second ESP application and passed them through a model. We experimented with all six MobileNets models, running each image through each model, one at a time. Each model had an input shape of $96 \times 96 \times 3$, so we reshaped each image into a $96 \times 96 \times 3$ image and then converted it to a tensor for the inference task. Once the inference was performed, the image was displayed on the screen, along with the probability of occurrence of each class.

III. Experimental Methodology

We used the setup described in Section II to evaluate the two settings – running the task locally on the microcontroller versus offloading it to the smartphone.

Accuracy: We performed the accuracy evaluation using an offline dataset of 1250 images (250 images each of *bicycle*, *car*, *chair*, *person*, and *table* classes) that were taken from multiple Kaggle datasets. MobileNets models use the parameter α to control the width of the network; we experimented with various α values to determine the object detection accuracy. For each model, we created both a quantized model (8-bit int values) for weights and biases and an unquantized model (32-bit floating point numbers) for weights and biases. The accuracy was estimated by performing 5-fold cross-validation.

Latency: The ESP device collected images using the onboard camera module for 30 seconds. We calculated the latency numbers for capturing the image, processing the image (either locally in application 1 or offloaded by sending it over Wi-Fi and then processing it in application 2) and producing the final output. We ran all the models using either app 1 (inference on the microcontroller) or app 2 (inference on a smartphone).

Energy: We computed energy from the microcontroller's current (or power) draw. We used the Monsoon High Voltage Power monitor device to compute the draw [7]. We performed various combinations of keeping the MCU, the camera, or RF modules either ON or OFF, choosing between a quantized and unquantized model, and choosing between performing the inferences locally, on the ESP device, or offloading the computation to the smartphone. To measure the power consumption when only the MCU was ON, we ran an empty loop() function. Subsequently, for turning components ON, we gradually included additional code into this subroutine.

IV. Results

A. Accuracy

Overall, for all models, we observed that the performance improves as the value of α increases. For example, in MobileNetv1's quantized models, the accuracy increased from 83.4% for $\alpha = 0.1$ to 89.9% for $\alpha = 0.2$. Regarding quantization, we observed that the performance of all quantized models was slightly lower than the unquantized models in all cases. Fig. 1 presents the accuracy of various models.

B. Latency

Overall, it takes 84.5 ms for the camera to capture and store an image in the RAM. The image capturing and inference required 415 ms for the unquantized model, whereas it required 135 ms for the quantized model (using $\alpha = 0.1$). We also observed that quantized models could complete inference at least 2$\times$ faster than unquantized models. For offloading the models, the inferencing time was comparable to on-device computation in the case of smaller models. However, the gap widened for larger models (higher values of α). If we consider the network latency, running the task on the smartphone is 2-3$\times$ faster than running the unquantized model on the ESP. Fig. 2 presents the variation of latency for different models.

C. Energy

We collected the current consumption details under various scenarios. Major results are presented in Table I.

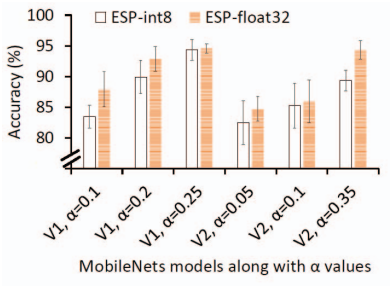

Fig. 1: Object detection of quantized and unquantized models trained with different α.

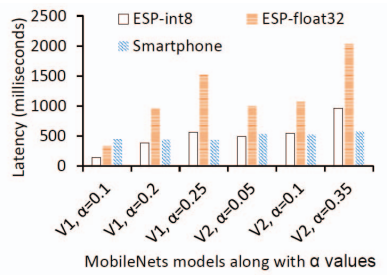

Fig. 2: Latency in prediction for various values of α, using both MobileNets V1 and V2.

Components ON	Setting	Curr. (mA)
MCU	none	44.84
MCU, cam.	frame: 0.2 fps	78.04
MCU, cam.	frame rate: 1 fps	79.15
MCU, cam., inf.	1fps, unquant.	87.18
MCU, cam., inf.	1fps, quant.	80.51
MCU, WiFi	tx rate: 1 kbps	139.45
MCU, WiFi	tx rate: 5 kbps	139.69
MCU, cam., WiFi	1fps	176.31

TABLE I: Current draw during operation of specific ESP components at various operation specifications

MCU ON: We observed that the average power draw when the MCU was ON was 340.1 mW, and the average current draw was 68.0 mA. The average current draw dropped to 44.8 mA (power draw of 224.2 mW) when a 1 ms sleep was introduced in the loop() function.

MCU + Camera: The average current (and power) draw for image captured at 2 fps, 1 fps, 0.66 fps, and 0.2 fps was 80.7 mA (403.3 mW), 79.2 mA (395.7 mW), 79.0 mA (394.9 mW), and 78.0 mA (390.2 mW) respectively. These readings show that reducing the frame rate does not substantially reduce the energy draw.

Camera + on Device inference using unquantized model: The average current (and power) draw for image capture at 2 fps, 1 fps, 0.66 fps, and 0.2 fps was 97.8 mA (489.0 mW), 87.2 mA (435.9 mW), 84.0 mA (420.2 mW), and 79.4 mA (396.7 mW) respectively.

Camera + on Device inference using quantized model: The average current (and power) draw for images captured at 2 fps, 1 fps, 0.66 fps, and 0.2 fps was 84.0 mA (419.9 mW), 80.5 mA (402.5 mW), 79.3 mA (396.4 mW), and 78.2 mA (390.9 mW) respectively. These readings indicate that at 2 fps, the current draw of the quantized model was substantially lower than that of the unquantized model.

MCU + Camera + WiFi: When the radio is ON and is waiting for a client station to join, the current draw is 146.3 mA. When the client is continuously transmitting 1-KiB and 5-KiB of data every second, the current draw is 139.5 mA, and 139.7 mA, respectively. Finally, when images were sent to the connected smartphone via Wi-Fi for inference purposes at 1 fps, the current draw was 176.3 mA, approximately 2$\times$ more than running inferences in situ.

V. Conclusion

This paper compares the performance of running deep learning inferences on a microcontroller versus offloading the inference to a nearby device. Overall, we observed that offloading the inference to a nearby device is energy expensive (2$\times$) while being slightly more accurate. However, the smartphone could provide the inference with low latency for larger models. Thus, there is a trade-off between energy, latency, and accuracy. We believe that an adaptive pipeline for object detection tasks could help balance system performance.

VI. Acknowledgment

This research results from a research program supported by BITS Pilani ACG GOA/ACG/2021-2022/Nov/05 and a gift from the TensorFlow team to develop educational resources. All findings and recommendations are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A survey of deep learning-based object detection," *IEEE access*, vol. 7, pp. 128 837–128 868, 2019.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[3] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals linear bottlenecks," in *Conference on Computer Vision & Pattern Recognition (CVPR)*, 2018.

[4] "ESP-EYE," https://www.espressif.com/en/products/devkits/esp-eye/overview, 2023, Accessed:02/13/2023.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] "Tensorflow hub," https://tfhub.dev/s?module-type=image-classification&network-architecture=mobilenet-v2, 2023, accessed:01/21/2023.

[7] "High voltage power monitor," https://www.msoon.com/high-voltage-power-monitor, 2023, accessed:01/26/2023.